Research Paper | Computer Science | E-ISSN No : 2454-9916 | Volume : 4 | Issue : 3 | March 2018

EXPLORING KOTLIN'S ENHANCEMENTS FOR MULTIPLATFORM PROJECTS

P. V. Kulkarni [1] | Mayur K. Jadhav [2]

[1] Assistant Professor, Computer Science & Engineering Department, Government College of Engineering, Aurangabad, India - 431005.

[2] ME Student, Computer Science & Engineering Department, Government College of Engineering, Aurangabad, India - 431005.

ABSTRACT

There has been a huge growing interest for using the newly launched Kotlin programming language in scientific and engineering applications. While Java which is one of the most widely used programming languages and also the official language of Android development, there are various reasons why Java might not always be the best option for various projects. This is just because Kotlin truly offers several features, which other traditional languages (C, C++, Java and FORTRAN) lack. Also, the historic poor performance of Java stops it from being widely used in scientific applications. Kotlin 1.2 was a major new release and a big step on the road towards enabling the use of Kotlin across all components of a modern application and multiplatform project. This study explores the major improvements and feature scopes over the year of the newly released enhanced version of Kotlin.

KEYWORDS: Kotlin, Programming language, Multiplatform Projects, Java, JavaScript, Android.

INTRODUCTION:

Kotlin is new software development language running on Java Virtual Machine, Android or browser. It is statically typed language and it can be compiled to JavaScript source code (running on browsers). Kotlin is brief, considerably reduced code boilerplate, safe, impossible to get an *NullPointerException*, versatile, flexible, general-purpose programming language, interoperable, can also be used with existing JVM libs and frameworks. A team at JetBrains (creators of IntelliJ Idea) developed Kotlin, an OSS language with an army of external contributors.

he interest in Kotlin programming language will always be increasing because day-by-day it is moving towards becoming the universal programming language. What is a universal programming language? From a unsophisticated view, the prospect could be that one language is used for all categories of programming. While that may be far-fetched in today's complex multifaceted world, the expectation could be accustomed to single language becoming the dominant programming language. Most certainly, it is the solitary, most important language to master. [2][3]

It's true that Java was one of the top programming languages, but time flies by real soon. In the up-to-the-minute modern world, no coder can imagine coding without the integration or support of lambda functions & streams, and no programmer demands to bother with plugins to accomplish the same. The Java I used on Android doesn't even have support for lambdas, method references, streams, try-with-resources (*minSdk ≥ 19*). Programmers still have to use the *javax.time* APIs from the ancient Java 6/7 worlds. One of the prime flaws in Java is the way it knobs the "null," leading to the feared *NulPointerException* (NPE), widely known as The Billion Dollar Mistake. [4][5]

This paper explores and adds significant knowledge to the field of Kotlin Programming by conducting an extensive literature review of the research done in the past and attempts to understand the changes included in the latest released Kotlin 1.2.30 as well as attempts to understand the scope in development work for multiplatform projects.

Hindrances of Kotlin 1.0:

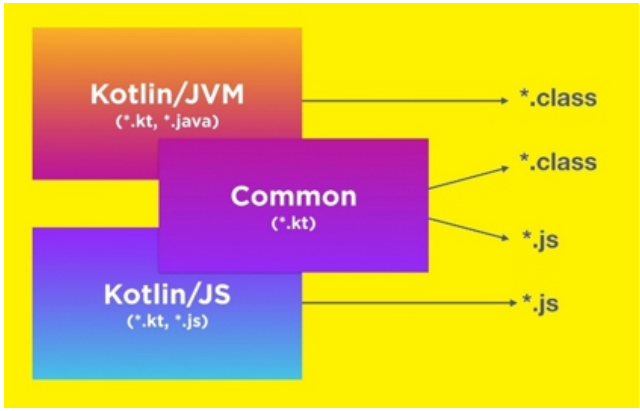
Intellij plugin for Kotlin's initial release was extremely buggy and very far behind Java. That time, Kotlin seemed to have promise a better Java, deprived of compromises. Well, it did achieve that at the language level. However, a big part of using Java was Jetbrains' own brilliant Java tooling. Kotlin's collection api had no correspondent to Java's *parallelStream()*. This was profoundly missed by developers. It was unable to subclass the Data classes in initial release of Kotlin 1.0. Here the Kotlin plugin was having a lot of effortful work to do in demand to catch up with what Jetbrains has built over 15 years for Java. The editor recurrently stopped doing the syntax highlighting, code completion etc. The only temporary solution when this occurred was to start over the entire IDE. The *Call Hierarchy* view recurrently was unable to show all calls to a method if that method was used athwart both Java and Kotlin. This was a super serious bug. The Kotlin *Gradle* integration was almost flaky in the past. Other than this, programmers were unable to see inferred variable type, *Refactoring* restrictions were present, No *Postfix Completion*, No *Duplicate Detection* support was present. [1]

Kotlin 1.1:

Kotlin's new version 1.1 was then released introducing a number of new language features most markedly *Coroutines* and improved support for its JavaScript target. All language features were supported, and there were many new tools for integration with the front-end development environment. Even though it was still considered experimental, one of the key new features in Kotlin 1.1 the *Coroutines*, which were available through the use of three higher-level constructs: *async*, *await*, and *yield*. But being experimental, *Coroutines* were opt-in only and there was an anxiety among programmers that the *Coroutines* API may change in forthcoming releases. Other significant new features added to Kotlin 1.1 were, *Type aliases* (which allowed programmers to define an alternative name for a type), The :: operator (to get a member reference to a method of specific object), Data classes could now be extended, Destructuring support in lambdas was also included.[6][8]

Multiplatform Projects:

In Kotlin' major release 1.1, support for multiplatform projects was included. A multiplatform project allows an individual to build multiple tiers of his/her application – backend, frontend and Android app – from the same codebase. Such a project contains both common modules, which contain platform-independent code, as well as platform-specific modules, which contain code for a specific platform (JVM or JS) and can use platform-specific libraries. To call platform-specific code from a common module, an individual can specify expected declarations – declarations for which all platform-specific modules need to provide actual implementations.



Note that multiplatform projects are presently an experimental feature; it means that the feature is ready for use, but the developers may need to change the design in the subsequent release (and if they do, they'll surely provide migration tools for existing code) allowing programmers to reuse code between target platforms supported by Kotlin – JVM, JavaScript and (in the future) Native. In a multiplatform project, programmers have three kinds of modules:

A common module contains code that is not specific to any platform, as well as declarations without implementation of platform-dependent APIs. [7]

A platform module contains implementations of platform-dependent declarations in the common module for a specific platform, as well as other platform-dependent code.

A regular module targets a specific platform and can either be a dependency of platform modules or depend on platform modules.

While compiling a multiplatform project for a specific platform, the code for both the common and platform-specific parts is generated. A key feature of the multiplatform project support is the possibility to express dependencies of common code on platform-specific parts through expected and actual declarations. An expected declaration specifies an API (class, interface, annotation, top-level declaration etc.). An actual declaration is either a platform-dependent implementation of the API or a *typealias* referring to an existing implementation of the API in an external library. Here's an example:

In common code:

```
// expected platform-specific API:
expect fun hello(world: String): String
fun greet() {
// usage of the expected API:
val greeting = hello("multi-platform world")
println(greeting)
}
expect class URL(spec: String) {
open fun getHost(): String
open fun getPath(): String
}
```

In JVM platform code:

```
actual fun hello(world: String): String =
"Hello, $world, on the JVM platform!"
// using existing platform-specific implementation:
actual typealias URL = java.net.URL
```

Enhancement in Kotlin 1.2.x:
While, in Kotlin 1.1, Intellij officially released the JavaScript target, permitting programmers to compile Kotlin code to JS and to run it in the browser, in Kotlin 1.2, they've added the possibility to reuse code between the JVM and JavaScript. Now programmers can write the business logic of their application once, and reuse it across all tiers of their application – the backend, the browser frontend and the Android mobile app. They've also operated on libraries to service programmers to reuse more of the code, such as a cross-platform serialization library. The release of Kotlin 1.2.30, was along with a new bugfix and tooling update for Kotlin 1.2.x. [9]

This update:

- Added a new-fangled declaration in the standard library, which imitates the suspend modifier for lambda expressions.
- Added provision for *TestNG* in *kotlin.test.*
- Brought provision for Android modules in multiplatform projects.
- Introduced a new feature in *kapt* for reporting annotation processing errors along with proper links to the original Kotlin declarations.
- Added a lot of new inspections and intentions in the IntelliJ plugin and progresses its performance.
- Fixed bugs in the compiler and IntelliJ plugin.

Changes in the libraries:
This release adds a innovative function to the Kotlin standard library with the following signature:

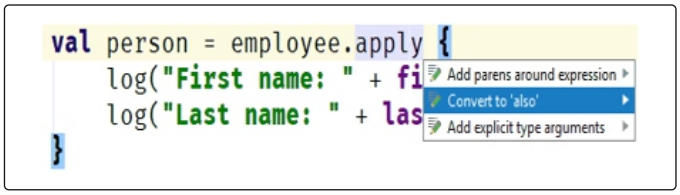
```
public inline fun <R> suspend(
noinline block: suspend () -> R
): suspend () -> R = block
```

The determination of this function is to wrap a function literal into a value of a suspending function type and enable its usage as a suspending function. Example:

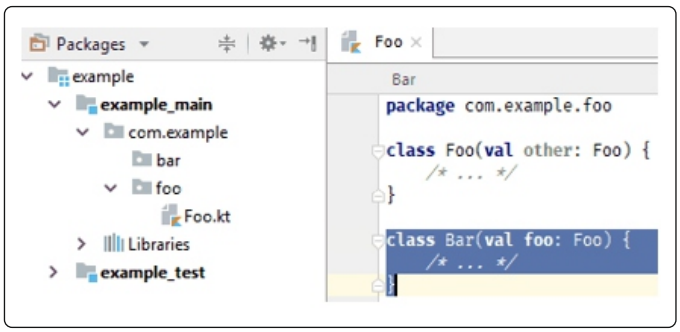
```
suspend {
val result = deferredResult.await()
renderResult()
}.startCoroutine(completion)
```

IntelliJ IDEA plugin enhancements:
This release brought various enhancements in the IntelliJ IDEA Kotlin plugin, such as performance enhancements, bug fixes, and new inspections and intentions. Intentions for converting the scoping function calls:

Kotlin 1.2.30 adds new intentions that convert calls to the scoping functions let and run and into each other, as well as also into apply and vice versa:



Pasting Kotlin code into a package:
The IntelliJ plugin now allows pasting Kotlin code into a package item in the Project View, creating a new Kotlin file for the code:



Other changes in the IDE plugin:
Data flow analysis ('Analyze Data Flow …') support for mixed Kotlin and Java codebases.

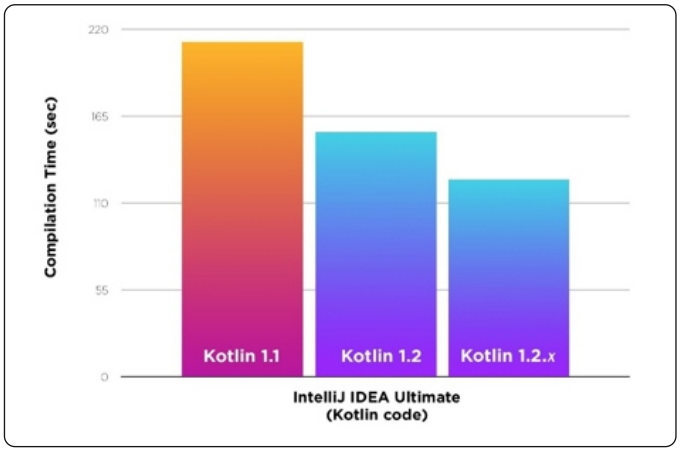
An option to create a run configuration for a Node CLI application from a main function in Kotlin/JS projects.

Enhancements in the Rename/Move refactoring, such as warnings on possible conflicts introduced by renaming. [9]

Changes in the compiler:
The Kotlin 1.2.30 update fixes several known issues in the Kotlin compiler and includes performance improvements.

The compiler is now able to optimize a tail call made in a suspending function to another Unit-returning suspending function, resulting into more efficient compiled code. [9]

Compilation Performance:
Over the course of development of 1.2, they've put a lot of effort in making the compilation process faster. The official developers have already reached approximately 25% improvement over Kotlin 1.1, and see significant potential for further improvements, for the releases in 1.2.x updates. The graph below shows the difference in compilation times for two large JetBrains projects built with Kotlin:



CONCLUSIONS:

In the noteworthy facets of this JVM language, one can express of its ease of project setup and its excellent Java interoperability. Taking the whole shebang into account, a good Android developer, got to give Kotlin a try. Its automated syntax converter that detects Java code as soon as you paste it in your current file, is a boon since it converts most of the Java code impeccably. Jetbrains has made a great job on tools and is incessantly extemporizing. To get started with it, it is therefore important to start introducing Kotlin with real and specific needs. Accompanied by full Java compatibility and good IDE support it is envisioned to expand code readability, which will give an easier way to encompass Android SDK classes and accelerate development.

REFERENCES:

1. M. K. Jadhav. Dripstst, M. (2016): Kotlin in Production-What works & What's broken, available at https://blog.dripstat.com/kotlin-in-production-the-good-the-bad-and-the-ugly-2/, accessed 18th March 2018.
2. Miloš Vasić. (2016): What is Kotlin?, in: Fundamental Kotlin, 1st Edition, Miloš Vasić, Belgrade, Serbia, pp. 6-7.
3. Stephen Samuel, Stefan Bocutiu. (2016): Getting started with Kotlin, in: Programming Kotlin, 1st Edition, Packt, Birmingham, UK, pp 3-4.
4. Manish Jangid. (2017). Kotlin-The unrivalled android programming language lineage. Imperial Journal of Interdisciplinary Research (IJIR), Vol-3, Issue-8, p. 256-259.
5. Prof. Ronak K. Panchal, Mr. Akshay K. Patel. (2017). A comparative study: Java Vs kotlin Programming in Android. International Journal of Innovative Trends in Engineering & Research, Vol-2, Issue 9, p. 4-10.
6. Sergio De Simone, S. (2017): Kotlin 1.1 Adds Coroutines, Type Aliases, Improved JavaScript Support, available at https://www.infoq.com/news/2017/03/kotlin-11-released, accessed 18th March 2018.
7. Dmitry Jemerov, D. (2017): Kotlin 1.2 Released: Sharing Code between Platforms, available at https://blog.jetbrains.com/kotlin, accessed 18th March 2018.
8. Mikhail Glukhikh, M. (2017): Kotlin 1.1 Release Candidate is Here, available at https://blog.jetbrains.com/kotlin/, accessed 18th March 2018.
9. Sergey Igushkin, S. (2018): Kotlin 1.2.30 is out, available at https://blog.jetbrains.com/kotlin/, accessed 18th March 2018.